京都大学学術情報リポジトリ
KURENAI 紅
Kyoto University Research Information Repository

京都大学
KYOTO UNIVERSITY

Title	可視化プログラミングの基礎(3) : GPUを利用した粒子ベースボリュームレンダリングの高速化
Author(s)	坂本, 尚久; 小山田, 耕二
Citation	計算工学 (2009), 14(2): 2060-2067
Issue Date	2009-04
URL	http://hdl.handle.net/2433/153377
Right	日本計算工学会
Type	Journal Article
Textversion	publisher

Kyoto University

チュートリアル

最近では、情報爆発時代における不可欠な技術として情報可視化技術の開発や先進的利用が注目されています。今回は、高度な可視化技術として位置づけられる有限要素法向けボリュームレンダリングに関するチュートリアル（全4回シリーズ）の第3回目として、京都大学の坂本 尚久先生と小山田 耕二先生に解説していただきます。

可視化プログラミングの基礎（3）
GPUを利用した粒子ベースボリュームレンダリングの高速化

坂本 尚久
小山田 耕二

1　はじめに

ボリュームデータを対象としたリアルタイム可視化技術は、科学技術可視化分野において主要な研究テーマの一つとなっており、近年ではGPU（Graphics Processing Unit）を利用したさまざまな高速可視化手法が提案されている。特に、節点が各軸上に等間隔に配置されるような規則格子ボリュームデータについては、GPU上のメモリ（テクスチャ）へ直接データを格納することができるため、格子間の補間計算やアルファ合成計算などをGPU上で高速に処理することが可能である[1]。一方、有限要素解析等で用いられることの多い非構造型ボリュームデータについては、その構造の複雑さから直接GPU上に格納することが困難であるため、要素ごとに半透明ポリゴンを重ね合わせてレンダリングすることで効率よくGPU上で処理する方法など[2]が提案されているが、アルファ合成計算を正しく行うためには、半透明ポリゴンに対する視点からの順序づけ計算（visibility sorting）が必須となる。この半透明ポリゴンの順序づけ計算では、すべてのポリゴンをGPUメモリに格納しておかなければならず、拡張性の観点で問題となる。特に大規模モデルの分散計算結果については格納そのものが不可能となる場合が出てくる。粒子ベースボリュームレンダリング（PBVR）では、この順序づけ計算は必要なく、かつ、ボリュームデータを微小粒子として表現することによってGPUの高速化機能を効率よく適用することが可能である。

本チュートリアルでは、最初にGPUプログラミングの基礎的な概要を説明したあと、PBVRの高速化についてアンプルコードを示しながら解説する。

2　GPU概要

一般に、GPU上では、3次元データを2次元画像に変換する一連の処理がパイプライン方式で実行されている。最近では、そのGPUパイプラインを構成するいくつかの処理を、ユーザが自由に処理を書き換えることができるプログラマブルシェーダ機構が提供されている。

2.1　レンダリングパイプライン

GPUは、3次元データを2次元画像へ変換する作業を並列処理によって効果的に実行できるよう設計されている。GPUを利用して描画を行うグラフィックスアプリケーションでは、この一連の変換処理が、いくつかの処理ステージに分割され、ひとつのパイプライン（レンダリングパイプライン）として固定的な手順で実行される（図1）。

一般的に、GPU上で実行されるレンダリングパイプラインでは、対象となる3次元データを構成する頂点データ（座標値、色、法線ベクトルなどの情報を含む）を、OpenGLやDirect3Dに代表されるグラフィックスAPI（Application Programming Interface）を利用してGPUに転送し、以下に述べる大きく4つの処理ステージを経て最終画像をディスプレイに表示する。

(1) ジオメトリ処理

ジオメトリ処理ステージでは、GPUに転送されてきた頂点データに対して、回転・平行移動・縮小/拡大などを行う。この処理ステージでは、頂点データが、オ

筆者紹介

さかもと なおひさ
平成13年（株）ケイ・ジー・ティー入社。平成19年京大大学院工学研究科博士課程了。平成20年京都大学特定助教。情報可視化に関する研究に従事。博士（工学）。電子情報通信学会、可視化情報学会各会員。

こやまだ こうじ
昭和60年京大大学院工学研究科了。同年日本IBM入社。平成10年岩手県立大学助教授、平成13年京都大学助教授、平成15年同教授。博士（工学）。情報可視化、設計最適化の研究に従事。IEEE CS、日本シミュレーション学会、情報処理学会各員。

ブジェクト座標(ローカル座標)からワールド座標(基準座標)へ変換され、続いて、カメラ座標へ変換された後、最終的にスクリーン座標に変換される。そして、頂点データを、点、線または面(三角形)で表わされる描画要素(プリミティブ)としてまとめ、描画する領域に応じて不要な情報を破棄したのち、次のステージに渡される。

(2) ラスター処理

ラスター処理ステージでは、プリミティブをフラグメント集合に変換する。各フラグメントは、スクリーン上の画素に対応しており、ラスター走査によって幾何プリミティブの内部に定義される。

(3) フラグメント処理

フラグメント処理ステージでは、前ステージで特定されるフラグメントに対して、プリミティブの各頂点の属性から補間計算が行われ、最終的な画素値が計算される。また、テクスチャ参照による補間計算を行うことも可能である。

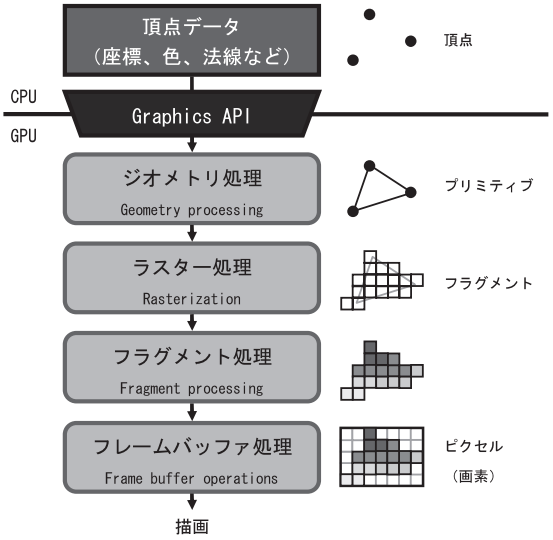


図1　レンダリングパイプライン

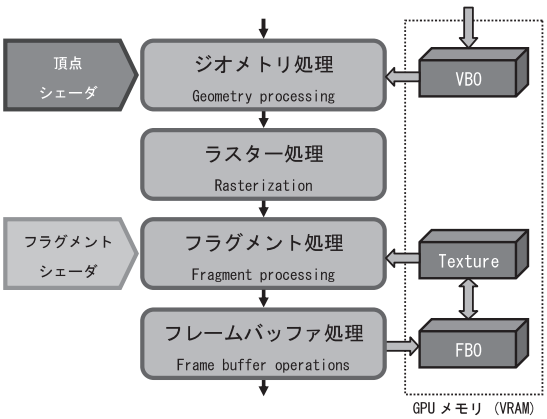


図2　プログラミングモデル

(4) フレームバッファ処理

フレームバッファ処理ステージでは、画素値が決定したフラグメントが、描画を行うための画像領域(フレームバッファ)に書き込まれる。このとき、対応する画素位置にすでにフラグメントが書き込まれていた場合、色の合成計算などを行うことも可能である。

2.2　プログラミングモデル

従来、GPUはグラフィックスデータを処理するための専用ハードウェアであったため、レンダリングパイプラインの各処理ステージの機能は固定化されており、利用者がそれを書き換えることは困難であった。しかし、近年では、GPU上のレンダリングパイプラインに対して利用者がプログラム可能な機構(プログラマブルシェーダ)が提供されており、それを利用することによって、従来時間がかかるとされていた高度な計算も、GPUの高速化機能を利用して高速で効果的なレンダリング計算を実現できるようになった。

現在のレンダリングパイプラインでは、頂点変換処理の書き換えを可能とする頂点シェーダ(vertex shader)とフラグメント処理の書き換えを可能とするフラグメントシェーダ(fragment shader)が利用できる(図2)。シェーダモデル4.0に対応したGPUにおいては、ジオメトリシェーダ(geometry shader)の利用も可能であるが、本稿での説明は省略する。

(1) 頂点シェーダ

頂点シェーダは、入力される頂点ごとに、変換方法のカスタマイズや頂点属性(座標値、色、法線ベクトルなど)の修正を行うために利用され、プログラム終了まで繰り返し処理される。

(2) フラグメントシェーダ

フラグメントシェーダは、入力されるフラグメントごとに、対応する画素の色や奥行き値の計算のために利用され、プログラム終了まで繰り返し処理される。

2.3　データの格納

シェーダを利用して高速なレンダリングを実現するためには、対象とするデータを格納するためのGPU上のメモリ領域が必要となる。最近のGPUであれば、テクスチャ、頂点バッファオブジェクト(Vertex Buffer Object, VBO)、フレームバッファオブジェクト(Frame Buffer Object, FBO)などが利用可能である。

(1) テクスチャ

テクスチャは、ポリゴンにテクスチャマッピングを行う際に、画素データを格納しておくための領域である。平面上に画素が規則正しく整列する(格子状に並ぶ)2次元テクスチャの場合、テクスチャ参照によって高速に格子間の値を2次線形補間(bi-linear interpolation)することが可能である。構造型ボリュームデータにおいては、そのデータを直接3次元テクスチャに格納するこ

とができ、ボリュームレンダリングを行う際の、格子間の補間計算（tri-linear interpolation）がGPU上で行えるため、リアルタイムで可視化を行うことも可能となる。

(2) 頂点バッファオブジェクト（VBO）

頂点バッファオブジェクトは、頂点データ配列をまとめてGPU上に格納するための領域である。頂点データを適切にGPU上に格納することによって、メインメモリからGPUへのデータ転送を少なくすることができ、効率よく処理を行うことが可能である。

(3) フレームバッファオブジェクト（FBO）

フレームバッファオブジェクトは、あらかじめ定義されている表示ウィンドウ向けのフレームバッファとは別に、2次元配列データを格納することができる。テクスチャと関連づけて利用する場合、FBOを利用することによってテクスチャへのレンダリングが可能となる。

3 粒子ベースボリュームレンダリングの高速化

前回のチュートリアルで解説したとおり、PBVRでは、要素毎に粒子を生成し、粒子投影後、サブピクセル処理を行うことによって最終画素値を決定する。このとき、粒子生成手法が乱数を利用した確率論的アプローチであるために、生成画像の一意性を保つことができず、レンダリングのたびに異なる画像となる可能性があるが、サブピクセルレベルを増加させることにより、この差を小さくすることができ、画質を向上させることができる[3]。しかし、サブピクセルレベルの増加に伴い、必要となるフレームバッファのメモリサイズも増大する。そのため、メモリリソースの比較的小さいGPU環境での実装を考える場合、メモリサイズの増大は、PBVRにとってはレンダリング画質向上のボトルネックとなる可能性がある。

この問題を解決するために、メモリコストの少ない低サブピクセルレベルでレンダリングを行い、その結果画像を累積していき、最後に平均化処理（アンサンブル平均化処理）を行うことによって高画質化する方法[4][5]をGPU実装する（図3）。

本章では、前回説明したPBVRのGPU実装とアンサンブル平均化処理を組み合わせた手法の実装について詳しく解説する。本チュートリアル中のサンプルコードは、3Dグラフィックス APIにOpenGL、シェーディング言語にGLSL（OpenGL Shading Language）の利用を想定している。また、サンプルコード中の「gl」,「GL」で始まる関数（定数）は、OpenGLの組み込み関数（定数）であり、シェーダコード内の「gl_」で始まるクラスや「vec3」などのクラスはGLSLの組み込みクラスである。これらの組み込み関数などについては、説明を省略した。詳しくは、参考文献[6][7]を参照されたい。

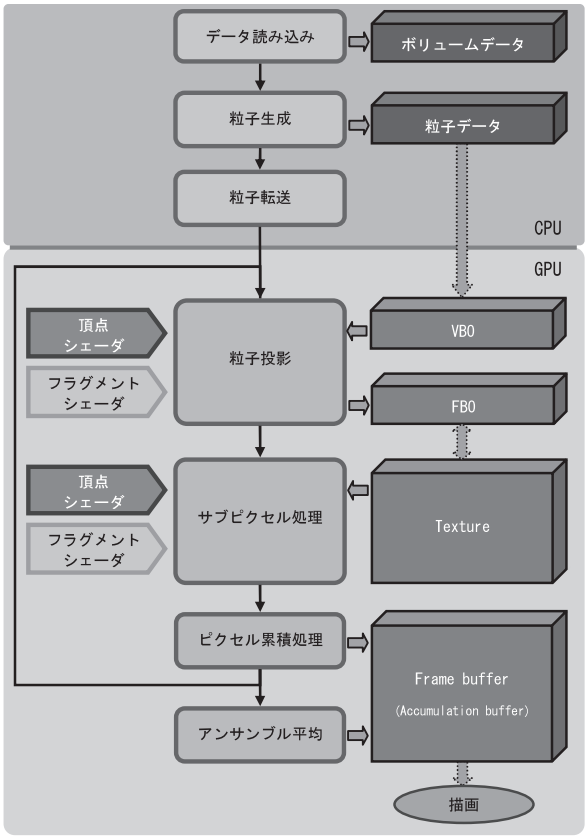


図3 GPUを利用した粒子ベースボリュームレンダリング

3.1 粒子データの転送

PBVRは、ボリュームデータから粒子を生成し、それらを単純に描画する手法である。PBVRをGPU上で実装するにあたり、粒子生成処理をGPUで行うことを想定すると、いったんボリュームデータをGPU上のメモリ領域（VRAM）上に格納する必要があり、メモリリソースの比較的少ないGPU環境においては、大規模データ可視化の足かせとなる可能性がある。そのため、CPU上で粒子生成処理を行い、生成された粒子群をGPUに転送しレンダリングを行う。粒子生成法は、前回のチュートリアルで解説をしたプログラムをそのまま利用する。生成された粒子は、座標値、色、法線ベクトルの情報を持っており、それらをGPUへ転送し、頂点バッファオブジェクト（VBO）に格納する。このとき、3.3節で詳述するアンサンブル平均化処理を効率よく実行するために、転送する粒子の並べ替えを事前に行っておく（図4）。粒子データ転送のサンプルコード（DownloadParticles）を以下に示す。

```
void DownloadParticles(
    PointObject& point, // ※並べ替えを事前に行っているものとする。
    VertexBufferObject* vbo )
{
    // 大域的に定義されるパラメータを取得する。
    float repetition_level = GetRepetitionLevle();

    // VBOを繰り返し（リピート）計算回数分用意する。
    vbo = new VertexBufferObject[ repetition_level ];

    // 各VBOに1リピート分ずつ粒子を転送する。
    for ( int r = 0; r < repetition_level; r++ )
```

```
    {
        // r番目のVBOに転送する粒子の個数を取得する。
        int n = GetNVerticesPerRpetition( r );

        // 粒子データ(座標、法線、色)のバイトサイズを計算する。
        size_t coord_size = sizeof(float) * n * 3;
        size_t normal_size = sizeof(float) * n * 3;
        size_t color_size = sizeof(unsigned char) * n * 3;

        // GPU上にr番目のVBO用メモリ領域を確保する。
        vbo[r].create( coord_size + normal_size + color_size );

        // 座標データを転送する。
        float* coord_ptr = point.coords().pointer() + n * 3;
        size_t coord_offset = 0;
        vbo[r].download( coord_size, coord_ptr, coord_offset );

        // 続けて、法線データを転送する。
        float* normal_ptr = point.normals().pointer() + n * 3;
        size_t normal_offset = coord_offset + coord_size;
        vbo[r].download( normal_size, normal_ptr, normal_offset );

        // 最後に、色データを転送する。
        unsigned char* color_ptr = point.colors().pointer() + n * 3;
        size_t color_offset = normal_offset + normal_size;
        vbo[r].download( color_size, color_ptr, color_offset );
    }
}
```

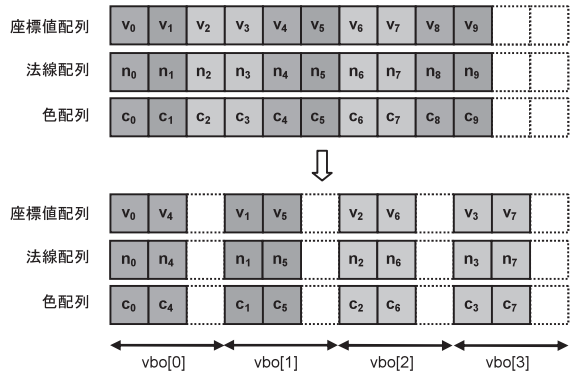


図4 粒子データの並べ替え(リピートレベル4の場合)

サンプルコード内のVertexBufferObjectクラスは、VBOに対する処理をサポートするクラスであり、GPU上にメモリを確保するメソッド(create)や、確保したメモリ領域にデータを転送するメソッド(download)などが実装されている。

3.2 粒子投影処理

VBOに格納されている粒子データを用いて投影計算を行う。このとき、投影された粒子の情報を格納するためのフレームバッファオブジェクト(FBO)を事前に準備しておく必要がある。粒子投影のためのFBOには、色データを格納するためのテクスチャと奥行き比較用のレンダーバッファ(デプスバッファ)を関連付けておく。FBOのセットアップ処理のサンプルコード(InitializeFBO)と粒子投影処理のサンプルコード(ProjectParticles)を以下に示す。

```
void InitializeFBO(
    FrameBufferObject* fbo,
    Texture2D* texture,
    RenderBuffer* buffer )
{
    //大域的に定義されるパラメータを取得する。
    float Ls = GetSubpixelLevel();
    int W = GetWindowWidth();
    int H = GetWindowHeight();

    // GPUメモリを初期化する。
    texture->release();
    buffer->release();

    // FBOを用意する。
    fbo->create();
    fbo->bind();

    // 色データ格納用のテクスチャを生成する。
    texture->release();
    texture->setPixelFormat( GL_RGB, GL_RGB, GL_UNSIGNED_BYTE );
    texture->create( W * Ls, H * Ls );

    // 奥行き比較用のレンダーバッファ(デプスバッファ)を生成する。
    buffer->release();
    buffer->setInternalFormat( GL_DEPTH_COMPONENT );
    buffer->create( W * Ls, H * Ls );

    // FBOに関連つける。
    fbo->attachColorTexture( texture->id() );
    fbo->attachDepthRenderBuffer( buffer->id() );
    fbo->bind();

    // 後処理
    texture->unbind();
    buffer->unbind();
    fbo->disable();
}
```

```
void ProjectParticles(
    VertexBufferObject& vbo,
    ProgramObject& shader
    )
{
    // 投影する粒子が格納されているVBOを指定する。
    vbo.bind();

    // シェーダ(頂点シェーダとフラグメントシェーダ)を利用するための
    // プログラムオブジェクトを指定する。
    shader.bind();

    // VBOに格納されている粒子の個数を取得する。
    size_t nvertices =GetNVertices( vbo );

// VBOに格納された粒子データ(座標、法線、色)に対する
// オフセットを取得する。
size_t coord_offset = GetCoordOffset( vbo );
size_t normal_offset = GetNormalOffset( vbo );
size_t color_offset = GetColorOffset( vbo );

    // 粒子を投影する。
    glEnableClientState( GL_VERTEX_ARRAY );
    glVertexPointer( 3, GL_FLOAT, 0, (char*)(coord_offset) );
    glEnableClientState( GL_NORMAL_ARRAY );
    glNormalPointer( 3, GL_FLOAT, 0, (char*)(normal_offset) );
    glEnableClientState( GL_COLOR_ARRAY );
    glColorPointer( 3,GL_UNSIGNED_BYTE,0,(char*)(color_offset) );

    glDrawArray( GL_POINTS, 0, nvertices );

    glDisableClientState( GL_VERTEX_ARRAY );
    glDisableClientState( GL_NORMAL_ARRAY );
    glDisableClientState( GL_COLOR_ARRAY );

    // シェーダの指定を解除する。
    shader.unbind();
}
```

前回のチュートリアルでは、粒子投影処理が終了した後に、サブピクセル処理を行いその段階でシェーディング計算を行っていた。今回のGPUを利用した粒子投影処理では、頂点シェーダを利用して投影計算を行い、粒子が投影されるテクスチャに対してフラグメントシェーダを利用してシェーディング計算を行うことで、サブピクセル値(サブピクセルの色)を計算する。シェーダを利用する際は、事前に、プログラムオ

ブジェクト（ProgramObject）を利用して、プログラム中でシェーダのコンパイルを行い、それぞれのシェーダの関連付けを行う必要がある。粒子投影処理用の頂点シェーダ（ProjectParticels.vert）とフラグメントシェーダ（ProjectParticles.frag）のサンプルコードを以下に示す。

```
// ファイル名：ProjectPartices.vert
varying vec3 position;
varying vec3 normal;

void main( void )
{
    gl_FrontColor = gl_Color;
    gl_Position = ftransform();
    normal = gl_Normal.xyz;
    position = gl_Position.xyz;
}
```

```
// ファイル名：ProjectPartices.frag
varying vec3 position;
varying vec3 normal;

void main( void )
{
    // Phongシェーディング
    vec3 L = normalize( gl_LightSource[0].position.xyz - position );
    vec3 N = normalize( gl_NormalMatrix * normal );
    float dd = max( dot( L, N ), 0.0 );

    vec3 ambient = vec3( 0.5, 0.5, 0.5 );
    vec3 diffuse = vec3( 0.5, 0.5, 0.5 );
    gl_FragColor.xyz = gl_Color.xyz * ( ambient + diffuse * dd );
    gl_FragColor.w = 1.0;
}
```

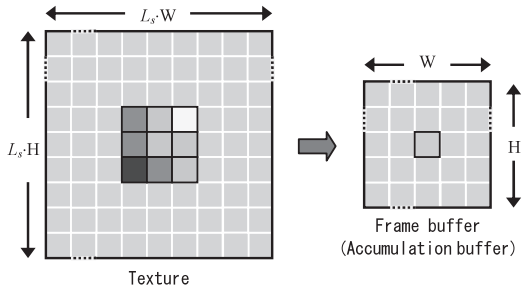


図5　サブピクセル処理（サブピクセルレベル3の場合）

3.3　サブピクセル処理

奥行き比較を有効にして粒子投影処理を行った結果、サブピクセル化されたテクスチャ領域に、粒子データ（シェーディング処理済み色データ）が格納される。GPUを利用したサブピクセル処理では、画像をサブピクセルレベルに応じて縮小することによって対応するフレームバッファ（アキュムレーションバッファ）の画素値を計算する（図5）。以下に、サブピクセル処理のサンプルコードを示す。

```
void SubpixelProcessing(
    Texture& texture,
    ProgramObject& shader
{
    // 大域的に定義されるパラメータを取得する。
    float Ls = GetSubpixelLevel();
    int W = GetWindowWidth();
    int H = GetWindowHeight();

    glMatrixMode( GL_MODELVIEW );
    glPushMatrix();
    glLoadIdentiry();
    glMatrixMode( GL_PROJECTION );
    glPushMatrix();
    glLoadIdentity();
    glOrtho( 0, 1, 0, 1, -1, 1 );
    {
        // テクスチャとシェーダを指定する。
        texture.bind();
        shaders.bind();

        // 縮小するためのパラメータを計算する。
        float step_x = 1.0f / ( W * Ls );
        float step_y = 1.0f / ( H * Ls );
        float start_x = -step_x * ( ( Ls - 1 ) * 0.5f );
        float start_y = -step_y * ( ( Ls - 1 ) * 0.5f );

        // 計算したパラメータをシェーダにセットする。
        shader.setUniformValuei( "texture", 0 );
        shader.setUniformValuef( "start", start_x, start_y );
        shader.setUniformValuef( "step", step_x, step_y );
        shader.setUniformValuef( "count", Ls, Ls );
        shader.setUniformValuef( "scale", 1.0f / ( Ls * Ls ) );

        // フレームバッファに書き込む（テクスチャマッピング）。
        glClear( GL_COLOR_BUFFER_BIT );
        glDisable( GL_DEPTH_TEST );
        glDisable( GL_LIGHTING );
        glBegin( GL_QUADS );
        glColor4f( 1.0, 1.0, 1.0, 1.0 );
        glTexCoord2f( 1, 1 ); glVertex2f( 1, 1 );
        glTexCoord2f( 0, 1 ); glVertex2f( 0, 1 );
        glTexCoord2f( 0, 0 ); glVertex2f( 0, 0 );
        glTexCoord2f( 1, 0 ); glVertex2f( 1, 0 );
        glEnd();
        glEnable( GL_LIGHTING );
        glEnable( GL_DEPTH_TEST );

        // テクスチャとシェーダの指定を解除する。
        shader.unbind();
        texture.unbind();
    }
    glPopMatrix();
    glMatrixMode( GL_MODELVIEW );
    glPopMatrix();
}
```

サブピクセル処理では画像の縮小を行うが、その際に、サブピクセルレベルに応じた平均化処理を行う。このとき利用する頂点シェーダ（SubpixelProcessing.vert）とフラグメントシェーダ（SubpixelProcessing.frag）のサンプルコードを以下に示す。

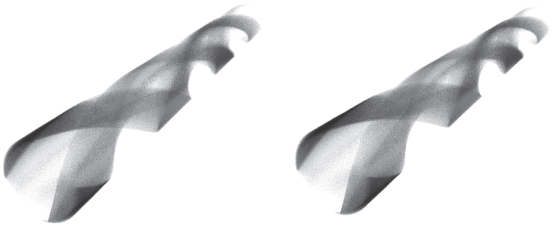
```
// ファイル名：SubpixelProcessing.vert
void main( void )
{
    gl_Position = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

```
// ファイル名：SubpixelProcessing.frag
uniform sampler2D texture;
uniform vec2 start;
uniform vec2 step;
uniform ivec2 count;
uniform float scale;

void main( void )
{
    int x, y;
    vec2 pos1, pos2, pos3;
    vec4 c = vec4( 0.0, 0.0, 0.0, 0.0 );

    pos1 = gl_TexCoord[0].st + start;
    float weight_xy, weight_y;
    for ( y = 0; y < count.t; y++ )
    {
        pos2 = pos1;
        if ( x + 1 < count.s )
        {
            weight_y = 2.0;
            pos2.t += step.t * 0.5;
```

```
            y++;
            pos1.t += step.t;
        }
        else
        {
            weight_y = 1.0;
        }

        for ( x = 0; x < count.s; x++ )
        {
            weight_xy = weight_y;
            pos3 = pos2;
            if ( x + 1 < count.s )
            {
                weight_xy = weight_y * 2.0;
                pos3.s += step.s * 0.5;
                x++;
                pos2.s += step.s;
            }
            c += texture2D( texture, pos3 ) * weight_xy;
            pos2.s += step.s;
        }
        pos1.t += step.t;
    }
    gl_FragColor = c * scale;
}
```

3.4　アンサンブル平均化処理によるピクセル値の決定

粒子投影を複数回行い、アンサンブル平均化処理を行うことにより、少ないメモリコストで画質を改善することができる。サブピクセル処理において、解像度W×Hの画像を生成する場合、サブピクセルレベルをL_sとすると、$(W \times L_s) \times (H \times L_s)$のテクスチャ領域が必要となる。このとき、$L_s \times L_s = L_s^2$個のサブピクセルを平均化することによって、対応する画素値(色)を計算する。一方、アンサンブル平均化処理では、サブピクセルレベルを1として、L_s^2回の粒子生成と粒子投影を行い、累積結果を平均化することによって、サブピクセルレベルL_sと同等の画像を得ることができる(図6)。このとき、粒子生成および粒子投影処理の繰り返し回数のことをリピートレベル(L_r)と呼ぶ。また、サブピクセル処理とアンサンブル平均化処理を組み合わせて計算することも可能であり、その場合サブピクセルレベル$L_s \cdot \sqrt{L_r}$相当の画像を得ることができる(図7)。



(a) サブピクセルレベル 10
リピートレベルは 1 に固定

(b) リピートレベル 100 (=10^2)
サブピクセルレベルは 1 に固定

図6　サブピクセル処理とアンサンブル平均化処理の比較
(ツイストドリルの応力解析結果、四面体素数：990万要素)

アンサンブル平均化処理では、指定されるリピートレベルに応じて粒子生成と投影を繰り返すことになる。しかし、GPU上で、粒子生成と投影を繰り返し行うと、メモリ領域の圧迫だけでなく、粒子生成にかかる計算コストが必要となり、リアルタイム処理のボトルネックとなる可能性がある。そのため、3.1節で説明したように、あらかじめ、粒子生成をCPU上で行っておき、生成された粒子をGPU上に格納することにする。ただし、サブピクセルレベルとリピートレベルが指定されたときは、レベル$L_s \cdot \sqrt{L_r}$で粒子生成を行う必要がある。さらに、リピートレベルに応じて繰り返し投影処理を行うために、図4に示す粒子の並べ替えを事前に行っておく必要がある。GPU上でのアンサンブル平均化処理の実装については、フレームバッファを構成するバッファの一つであるアキュムレーションバッファを利用する。サンプルコードを以下に示す。

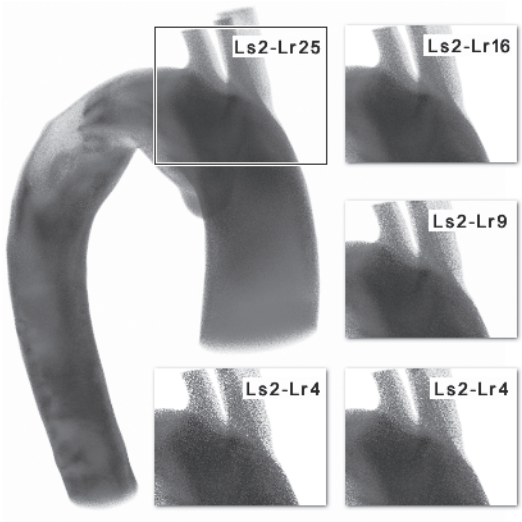
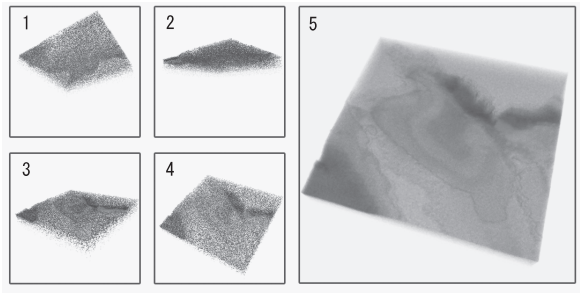


図7　サブピクセル処理とアンサンブル平均化処理の合成

```
void EnsembleAveraging(
    VertexBufferObject* vbo,
    ProgramObject& proj_shader,
    ProgramObject& subpix_shader,
    FrameBufferObject& fbo,
    Texture& texture )
{
    // 大域的に定義されているパラメータを取得する。
    size_t Ls = GetSubpixelLevel();
    size_t Lr = GetRepetitionLevel();
    int W = GetWindowWidth();
    int H = GetWindowHeight();

    // アキュムレーションバッファを初期化する。
    glClear( GL_ACCUM_BUFFER_BIT );

    // 累積計算
    for ( int r = 0; r < Lr; r++ )
    {
        // FBO(テクスチャ)に粒子を投影する。
        fbo.bind();
        glViewport( 0, 0, W * Ls, H * Ls );
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
        ProjectParticles( vbo[r], proj_shader );

        // サブピクセル処理を行い、結果をアキュムレーションバッファに
        // に書き込む。
        fbo.unbind();
        glViewport( 0, 0, W, H );
        SubpixelProcessing( texture, subpix_shader );

        // 累積する。
        glAccum( GL_ACCUM, 1.0f );
    }

    // 平均化する。
    glAccum( GL_RETURN, 1.0 / Lr );
}
```

アンサンブル平均を行うサンプルコードにおいて、入力となるproj_shaderおよびsubpix_shaderは、それぞれ、粒子投影処理およびサブピクセル処理を行うためのシェーダを管理するプログラムオブジェクトであるが、事前に以下のように初期化しておく必要がある。

```
void InitializeShaders(
    ProgramObject* proj_shader,
    ProgramObject* subpix_shader )
{
    // 粒子投影用頂点シェーダを読み込みコンパイルする。
    ShaderSource proj_vert_src;
    proj_vert_src.read( "ProjectParticles.vert" );
    VertexShader proj_vert_sh;
    proj_vert_sh.create( proj_vert_src );

    // 粒子投影用フラグメントシェーダを読み込みコンパイルする。
    ShaderSource proj_frag_src;
    proj_frag_src.read( "ProjectParticles.frag" );
    FragmentShader proj_frag_sh;
    proj_frag_sh.create( proj_frag_src );

    // 粒子投影用シェーダをリンクする。
    proj_shader->link( proj_vert_sh, proj_frag_sh );

    // サブピクセル処理用頂点シェーダを読み込みコンパイルする。
    ShaderSource subpix_vert_src;
    subpix_vert_src.read( "SubpixelProcessing.vert" );
    VertexShader subpix_vert_sh;
    subpix_vert_sh.create( subpix_vert_src );

    // サブピクセル処理用フラグメントシェーダを読み込みコンパイルする。
    ShaderSource subpix_frag_src;
    subpix_frag_src.read( "SubpixelProcessing.frag" );
    FragmentShader subpix_frag_sh;
    subpix_frag_sh.create( subpix_frag_src );

    // サブピクセル用シェーダをリンクする。
    subpix_shader->link( subpix_vert_sh, subpix_frag_sh );
}
```



(a) 操作中はリピートレベル 1 で粗くレンダリング（60fps）　(b) 静止時はリピートレベル 49 で高精細にレンダリング（9fps）

図8　詳細度（LOD）制御の例
（地震シミュレーションの結果、四面体要素数：270万要素）

3.5　詳細度の制御

前回のチュートリアルでも示した通り、PBVRでは、サブピクセルレベルを上げることにより画質が向上するが、それに伴いレンダリング速度が低下する。そのため、動作環境に応じてサブピクセルレベルを調整し、画像の詳細度（Level of Detail, LOD）を制御することによって効果的なレンダリングが可能となる。しかし、サブピクセルレベルの変更により、粒子の再生処理が要求されるため、対話的なLOD制御が困難である。それに対して、アンサンブル平均化処理では、累積計算を行う回数（繰り返し回数）を変更するだけで、レンダリング可能であるため、対話的なLOD制御が可能である。たとえば、サブピクセルレベルを1に固定して、リピートレベルを49としてレンダリングする場合、マウスでの操作中は繰り返し回数を1でレンダリングを行い、静止したところで繰り返し回数を49としてレンダリングを行うことによって対話性の高い効果的な可視化を行うことが可能である（図8）。

4　可視化システムの実装

本章では、前回同様、可視化基盤ライブラリKVSを利用して、GPU版PBVRを実装し、簡単な可視化システムを構築する。そして、構築したシステムを利用して、いくつかの非構造型ボリュームデータを可視化した結果を示す。前章で示したサンプルコードで利用したいくつかのクラスは、KVS上に実装されている。クラスの対応を表1に示す。

表1　クラスの対応

サンプルコード内で利用されているクラス	KVS上で実装されているクラス
VertexBufferObject	kvs::glew::VertexBufferObject
FrameBufferObject	kvs::glew::FrameBufferObject
Texture2D	kvs::Texture2D
RenderBuffer	kvs::glew::RenderBuffer
ProgramObject	kvs::glew::ProgramObject
ShaderSource	kvs::glew::ShaderSource
VertexShader	kvs::glew::VertexShader
FragmentShader	kvs::glew::FragmentsShader

4.1　レンダラの実装

GPU版PBVRをKVSのレンダラクラスとして実装することを検討する。前章に示したサンプルコードのうち、実行時一度だけ呼び出される初期化処理とレンダリングを行うときに毎フレームごとに呼び出される描画処理は以下のようにまとめることができる。

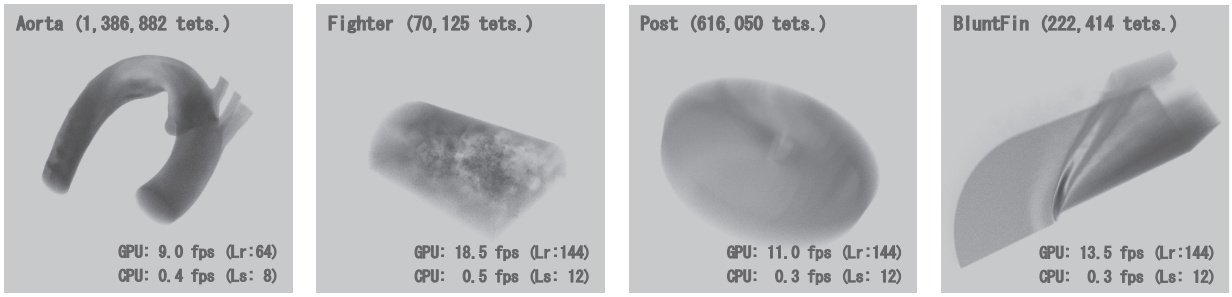
初期化処理

- 粒子データ転送（DownloadParticles）
- FBO初期化（InitializeFBO）
- シェーダ初期化（InitializeShaders）

描画処理

- アンサンブル平均化処理（EnsembleAveraging）
- 粒子投影（ProjectParticles）
- サブピクセル処理（SubpixelProcessing）

描画処理については、フレームごとに呼び出されるレンダラクラスのexecメソッド内に記述し、初期化処理は、execメソッドが呼び出されるより以前に1回だけ実行するように実装する。このレンダラクラスの正確な実装は、KVSのkvs::glew::ParticleVolumeRendererクラスを参照されたい。また、前回のチュートリアルで構



(a) 大動脈瘤の影響による血流シミュレーション結果　(b) ジェット戦闘機まわりの気流シミュレーション結果　(c) 液化酸素（非圧縮性流体）の流体シミュレーション結果　(d) 鈍頭フィンの設置板上の気流シミュレーション結果

図9　GPU版PBVRシステムを用いた可視化結果

築した可視化システムに対して以下のようなレンダリングモジュールを作成し、CPU版のPBVRレンダリングモジュールと入れ替えることによって簡単にGPU版PBVRシステムを構築することができる。

```
// ファイルの先頭に
//   #include <kvs/glew/GLEW>
//   #include <kvs/glew/ParticleVolumeRenderer>
// を追加する。

// サブピクセルレベル：subpixel_level
// リピートレベル：repetition_level
// ※粒子生成処理の際に指定するサブピクセルレベルは
//   subpixel_level * sqrt( repetition_level )

kvs::PipelineModule r( new kvs::glew::ParticleVolumeRenderer );
r.get<kvs::glew::ParticleVolumeRenderer>()
  ->setSubpixelLevel( subpixel_level );
r. get<kvs::glew::ParticleVolumeRenderer>()
  ->setRepetitionLevel( repetition_level );
```

4.2　実験

前節で述べたレンダラを利用して、Intel Core2 Duo 1.8GHz、2GB RAM、nVidia GeForce 8800 GTS（384MB）で構成されるコンピュータ上で、テストデータを用いてレンダリングパフォーマンスの計測を行った。結果を図9に示す。実験に利用したデータは、前回のチュートリアルの実験で用いたものと同じである。実験結果から、CPU版と比較してGPU版ではおよそ20〜40倍の高速化が確認できた。GPUで高速化されたPBVRを利用することによって、汎用的なコンピュータ上で非構造型ボリュームデータを対話的な速度で可視化することが可能である。

5　おわりに

本チュートリアルでは、GPUを利用した粒子ベースボリュームレンダリングの高速化について解説した。特に、粒子投影処理、サブピクセル処理、アンサンブル平均化処理について、シェーダコードを含むサンプルコードを示し具体的な実装方法について解説した。GPUを利用することで、画質を落とすことなく、大幅な速度向上を確認した。http://www.youtube.com/user/kyotoviz で実際にレンダリングしているデモ映像を確認することができる。

最終回となる次回は、大規模分散データの可視化にフォーカスをあて、粒子ベースボリュームレンダリングの並列処理およびタイルドディスプレイ装置を利用した高精細画像レンダリングについて解説を行う予定である。

■参考文献

[1] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, and D. Weiskopf, Real-time Volume Graphics, A. K. Peters, Ltd., (2006).

[2] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon, A survey of GPU-based volume rendering of unstructured grids. Brazilian Journal of Theoretic and Applied Computing (RITA), Vol.12, No.2, pp.9-29, (2005).

[3] 河村 拓馬, 坂本 尚久, 山崎 晃, 小山田 耕二, 粒子ベースボリュームレンダリングのための粒子密度推定法 -大規模非構造ボリュームデータに対する適用-, 可視化情報学会論文集, Vol.28, No.11, pp.69-77, (2008).

[4] K. Koyamada, N. Sakamoto, S. Tanaka, A Particle Modeling for Rendering Irregular Volumes, International Conference on Computer Modeling and Simulation (UKSIM 2008), pp.372-377, (2008).

[5] D. Zhongming, T. Kawamura, N. Sakamoto, K. Koyamada, GPU Acceleration of Improved Particle-based Volume Rendering for Irregular-grid Data, Proceedings of International Conference on System Simulation and Scientific Computing (ICSC 2008), pp.685-692, (2008).

[6] M. Woo, J. Neider, and T. Davis, OpenGL Programming Guide (Version 2), Addison-Wesley Pub., (2005).

[7] R. J. Rost, OpenGL Shading Language (2nd Edition), Addison-Wesley Pub., (2006).